Proof-of-concept prototypes

V. Cahill, A. Casimiro, J. Kaiser, P. Martins, V. Reynolds,
P. Sousa, P. Veríssimo and M. Wu

DI-FCUL TR–03–20

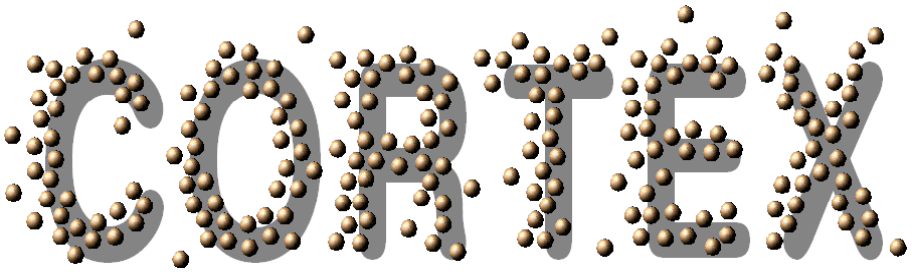
July 2003

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Project IST-2000-26031

CO-operating Real-time senTient objects: architecture and EXperimental evaluation



Proof-of-concept prototypes

CORTEX Deliverable D7

Version 1.0

April 30, 2003

Revisions

Rev.	Date	Comment
0.1	18/04/2003	Draft document for internal review
1.0	30/04/2003	Final document

Editor

António Casimiro, University of Lisboa

Contributors

Vinny Cahill, Trinity College Dublin
António Casimiro, University of Lisboa
Jörg Kaiser, University of Ulm
Pedro Martins, University of Lisboa
Vinny Reynolds, Trinity College Dublin
Paulo Sousa, University of Lisboa
Paulo Veríssimo, University of Lisboa
Maomao Wu, University of Lancaster

Address

Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande
1749-016 Lisboa
Portugal

Executive Summary

The main objective of this deliverable is to provide a set of proof-of-concept prototypes to illustrate some particular solutions developed within CORTEX. Additionally, these proof-of-concept prototypes should also be seen as technology demonstrators, which will provide relevant input for the construction of integrated CORTEX demo applications (see deliverable WP4-D8).

More specifically, this deliverable includes the following five proof-of-concept prototype demos:

- Cooperating Autonomous Robots Demo
- Adaptation and Fail-safety in Cooperating Cars Demo
- Sentient Vehicle Demo
- Sentient Room Demo
- Demo of Framework for Testing Safety-Critical Sentient Applications

The main issues focused by these demos include: a) communication through event-channels; b) the use of sensor technology; c) dealing with timeliness and fail-safety requirements; d) QoS adaptation; e) context awareness; f) sensor fusion; g) the use of a component framework. The present document essentially provides an overview of these demos, pointing out their objectives and describing their main steps by means of a storyboard.

Contents

1 Introduction

There are several challenges that must be addressed when programming real-time sentient objects and applications. However, as identified in deliverable WP1-D1: Definition of Application Scenarios, each application scenario stresses in different ways the problems that have to be addressed. Therefore, we believe that a correct approach to the demonstration and assessment of our results is to focus on different and representative applications in order to cover a larger number of issues.

In order to achieve this goal, and according to the work plan defined in the CORTEX technical annex, we need to demonstrate the feasibility and validity of the several solutions that have been proposed so far. This has been done through the construction of several proof-of-concept prototypes, each of them focusing and demonstrating one or more of these solutions. Given that we are also working towards the definition of the final application scenarios that will be used to demonstrate the several CORTEX concepts, these proof-of-concept prototypes have been defined and constructed having in mind the anticipated final applications, in order to allow an easier integration of the several solutions. The preliminary results concerning this integration are in fact presented in deliverable WP4-D8: Analysis and design of application scenarios.

Given the above, deliverable WP3-D7 is not a text deliverable. Instead, it consists of a set of proof-of-concept prototypes, which are demonstrable, and which include four demonstrators related to application scenarios and an additional demonstrator of a supporting tool for testing real-time sentient applications. More specifically, the proof-of-concept prototype demonstrators are the following:

Cooperating Autonomous Robots Demo: this demo focuses on coordination aspects, on the use of event-channels as a means to support coordination and on the use of sensor technology;

Adaptation and Fail-safety in Cooperating Cars Demo: this demo focuses on the problem of dealing with timeliness and safety requirements, using adaptation and event-based communication;

Sentient Vehicle Demo: this demo addresses the problems of context awareness, sensor fusion and adaptation, based on the information provided by a range of different sensors;

Sentient Room Demo: this demo focuses on the use of sensor technology associated to the programming of sentient object components and it illustrates the use of a component framework;

Demo of Framework for Testing Safety-Critical Sentient Applications: the objective of this demo is to highlight the capabilities of the testing framework, using a simulated scenario of moving objects.

The present document provides a brief report on the background of these proof-of-concept prototypes. It includes a summary of the prototypes rationale, structure, and purpose. We make it clear, however, that deliverable WP3-D7 consists of the actual proof-of-concept prototypes and not on this document.

2 Cooperating Autonomous Robots Demo

Technical Objectives:

- dynamic subscriptions to event channels;
- dynamic configuration and plug&play;
- coordination of a distributed heterogeneous sensor infrastructure;
- exploitation of remote sensors in local control.

Plot of the demo:

Starting from some initial state and position, the robots try to achieve a coordinated motion pattern (platoon). The experimental set-up guarantees that the coordinated action can be only achieved and maintained if the robots share the environment perception provided by their embedded sensors. As the robots progress from the initially uncoordinated to the coordinated state, their information needs will change and involve the sensor information from other robots. The application copes with the evolving context by dynamically cancelling subscriptions and subscribing to new specific event types. The context includes proximity and relative motion of robots.

Demonstration of technical achievements:

The demonstration will show the coordinated interaction of autonomous mobile robots exploiting the event channels of the publisher/subscriber protocol. The robots have different sensors, thus perceiving different aspects of the environment. The demonstration highlights that:

1. An individual mobile robot does not have the capability of assessing the complete environmental context just based on local sensors. Therefore robots must cooperate.
2. During the demonstration, the roles of the information producer and information consumer are dynamically changed. This will show the possibility to dynamically subscribe and unsubscribe to channels as a publisher or a subscriber. Selective information access is provided by the event middleware, by means of subscriptions and event filtering.
3. The activity of the robots and their internal state is continuously monitored by an external control station by subscribing to the respective event channels and visualized online. The supervisory system has also the possibility to send control commands to the robots. The monitor tracks internal system activities of the publisher/subscriber middleware as announcements, subscriptions, un-subscriptions, events, detection of new context and distributed state transitions.

3 Adaptation and Fail-safety in Cooperating Cars Demo

Context:

The car of the future will communicate with other cars or entities in its proximity, over a wireless link, to achieve cooperation and coordination in certain occasions (this is one of the scenarios described in WP1-D1). The demo described here targets a more specific scenario where cars will have to achieve coordination when approaching a cross with no traffic signs where the right-priority rule applies.

In this scenario, each car periodically transmits an event (with its position, speed and direction) to the other cars, so that each one maintains always a correct perception of the reality, i.e. a real-time image of the reality (let's call it RT-image).

To achieve this RT-image, events have to be timely delivered. Only in this way it is possible to secure that the RT-image is consistent (apart a bounded error) with the reality and that car crashes can be avoided. Intuitively, the delivery deadline of an event is related with the sender speed: a higher speed requires a smaller deadline. If a car detects a deadline violation on an event it should receive, it must (timely) enter a fail-safe state (e.g. stopping) because its RT-image may have become inconsistent. Moreover, when the environment is "slow", each car should adapt their speed (slowing down) to avoid the occurrence of missed deadlines (i.e. timing failures). Without this adaptation to the environment (i.e., to the QoS provided by the communication substrate), a speedy car in a degraded environment could cause all other cars in its proximity (i.e., that need to receive its events very quickly) to stop.

Description:

The demo unrolls in a city area where it only exists perpendicular streets. There are 3 cars (which we depict with a different color: red, green or blue), moving in a different street and always in the same direction (up, down, left or right). Figure 1 presents a snapshot of the streets and cars representation, in the demo.

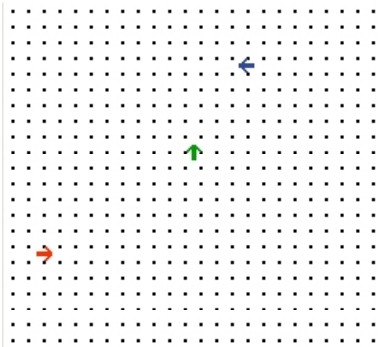


Figure 1: Snapshot of the demo's streets and cars. Red car is going right, green car is going up and blue car is going left.

All cars have the same control logic in each cross, advancing if the two following conditions are satisfied:

- there is no car in front;
- there is no car approaching from the right.

Each car has a position, speed and direction. All this information is periodically transmitted in the form of events to all other cars, such that each one can maintain a RT-image of other cars' position.

A car has also two optional features that are directly related with the goal of this demo:

Fail-Safety When this feature is ON, each event is marked with a deadline that depends on the sender speed. The delivery delay of events is measured using the Timely Computing Base (TCB) distributed timing failure detection service (the TCB model and its services were introduced in WP3-D4 and their implementation explained in WP3-D5) and if a timing failure is detected, a fail-safety handler that stops the car is timely executed;

QoS-Adaptation When this feature is ON, two operational modes can further be considered:

Coverage stability In this mode, cars adapt their speed according to the pair <*QoS allowed by the environment, desired coverage*>. More specifically, each car constructs a Probability Density Function (PDF) using the measurements provided by the TCB distributed measurements service, deriving a coverage (see *Dependable QoS Adaptation* section in WP2-D3) for each measurement (which is represented by a <*measurement, coverage*> function). Then, at each instant, the measurement associated with the desired coverage is used to calculate the (possibly new) car speed. Note that this conversion between *speed* and *measurement* entirely depends on the cars' logic;

Coverage awareness In this mode, at each instant, cars know the coverage associated with the pair <*QoS allowed by the environment, current speed*>. The PDF function described above is also used but now focusing on the coverage associated with the measurement that results (according to the cars' logic) from the current car speed.

Architecture of the demo:

In the demo, each car is represented by an IPAQ that simulates its position (in a real scenario, a location mechanism such as a GPS receiver would do this job), speed (changeable through the IPAQ's car interface) and direction (always the same). IPAQs communicate using IEEE 802.11b[1] in ad-hoc mode. Besides the IPAQs, there is a monitor application running in a laptop also using 802.11b ad-hoc to communicate with cars. The architecture is depicted in figure 2.

The monitor application is divided in three types of panels:

Reality View panel shows the actual reality allowing to observe cars behavior and therefore to detect car crashes (figure 1);

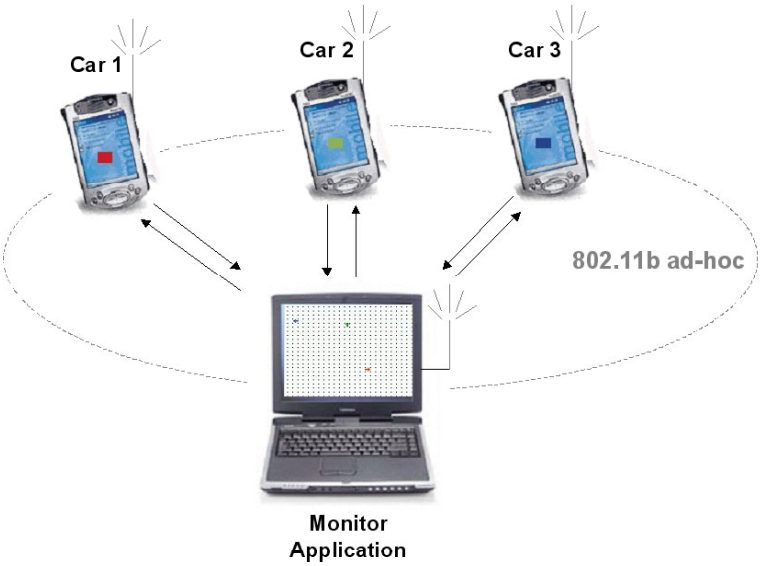


Figure 2: Demo architecture.

Master Control panel allows to turn ON/OFF certain car features, such as fail-safety and QoS-adaptation;

Car Control panel there exists one panel of this type per car. Each one has three parts:

Speed control allows to change car speed, increasing or decreasing it;

Statistics presents some car statistics: average delay of events received, number of events lost;

QoS-adaptation control allows to specify the parameters (e.g. coverage) of the QoS-adaptation feature and presents the <*measurement, coverage*> function described above.

Guideline for the demo:

The demo presentation is divided in 3 major parts:

- Enumeration of the various components;
- Fail-safety demo;
- QoS-adaptation demo.

Components

The cars interface and the various parts of the monitor application interface are explained.

Fail-Safety Demo

One extra feature of the monitor application is the possibility of injecting an electric blackout in a car's landscape, which we call "fog". When this fog is injected in a car, it stops seeing other cars approaching (i.e. all events sent by other cars are lost).

With the fail-safety feature OFF and if we inject fog in a car, car crashes are more likely to happen. We show a car crash happening in these conditions. Then we turn ON fail-safety and we see that the fog-injected car stops before any car crash occurs. This happens because when an event is lost, a timing failure occurs and, as described above, when the fail-safety feature is ON, a timing failure originates the shift to a fail-safe state (in this case, the car stops). When a car enters the fail-safe state, a siren appears in the monitor application's car control panel. A car can recover from a fail-safe state as soon as a timely event is received from all cars from which a late event was received. After the recovery, the siren disappears.

QoS-Adaptation Demo

Even without fog, when fail-safety feature is ON, cars obviously stop if they detect a timing failure. The probability of an event to suffer a timing failure is directly proportional to the sender speed. As mentioned above, in the context, a speedy car in a degraded environment could cause all other cars in its proximity (i.e. that should receive its events) to constantly detect timing failures and therefore to stop.

To avoid this, we turn ON QoS-adaptation in coverage stability mode. If we select a sufficiently high coverage (through a coverage slider that appears in each car control panel), cars adjust their speed to reflect the environment conditions and in this way the probability of a timing failure to occur decreases. Note that these environment conditions can be different from car to car because in this demo they are mapped into the PDF function constructed with the delays of events received from the other cars. These differences can be visually perceived by analyzing the $<$*measurement, coverage*$>$ function presented in each car control panel.

If, for a certain car, we switch to coverage awareness mode, the QoS-adaptation feature stops adapting its speed and starts informing about the coverage of the current speed. When the speed is increased, the coverage decreases, and when the speed is decreased, the coverage increases. Note that even if speed is not changed, coverage will change when environment conditions suffer modifications.

4 Sentient Vehicle Demo

Storyboard:

- The sentient vehicles sensors examine the road environment and upon determining it is safe to begin its journey, it does so (figure 3).

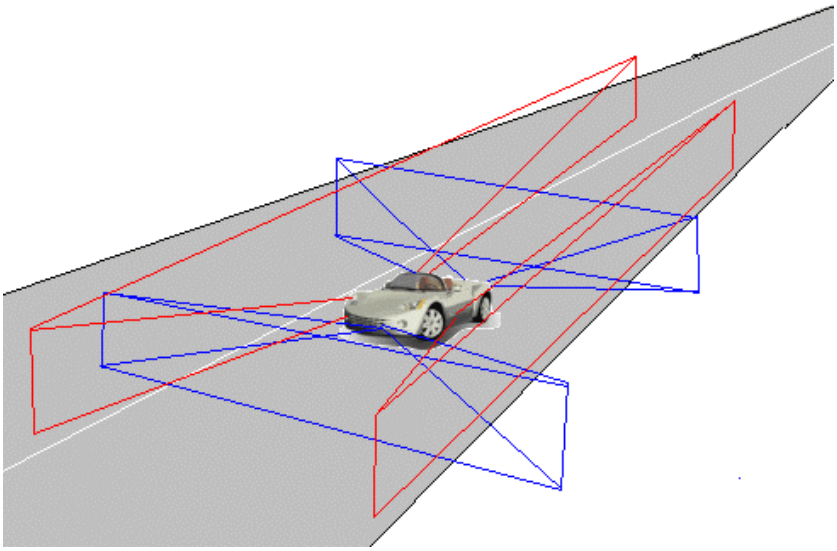


Figure 3:

- As it approaches its desired speed (the speed limit), its stereovision system detects another vehicle within its protected zone. The vehicles long range laser tells the vehicle that the detected vehicle is travelling at a slower speed (figure 4).

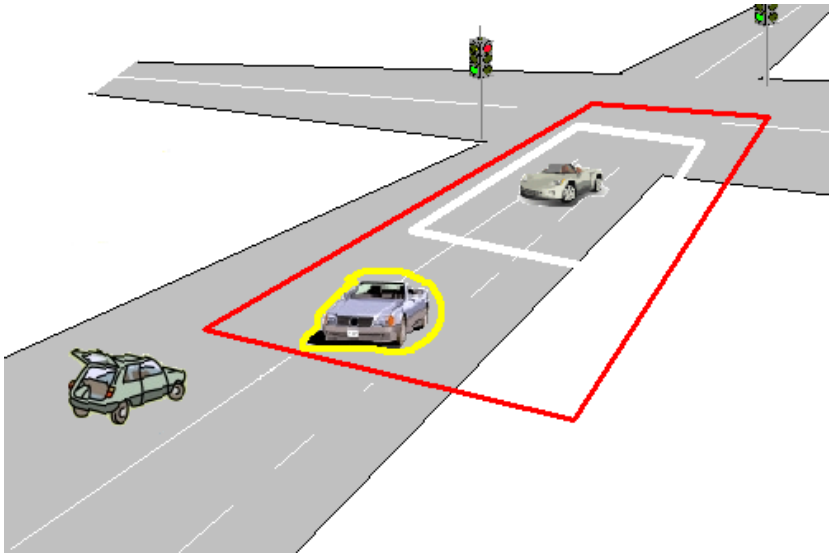


Figure 4:

- The vehicle cannot safely overtake because of the vehicle in the other lane and reduces its speed so that the detected vehicle remains outside of its safety envelope (figure 5).

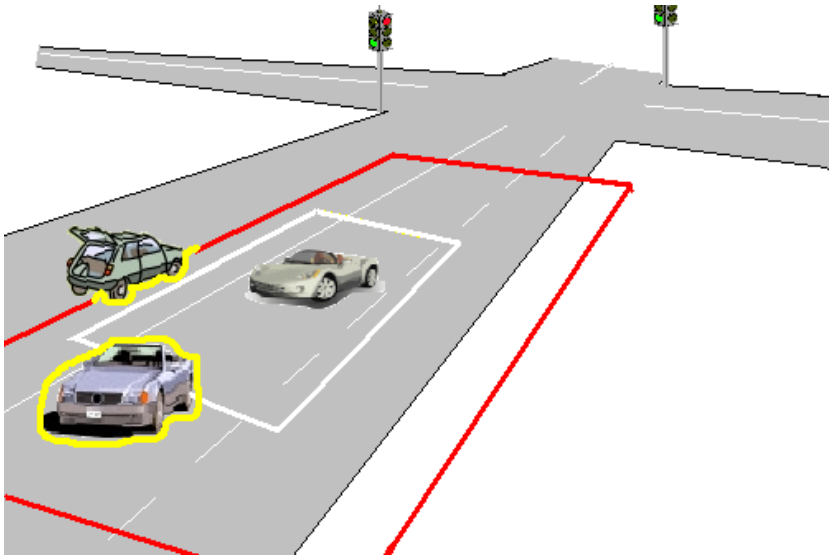


Figure 5:

- The vehicle establishes a communication channel with the blocking vehicle and requests it to change lane. The blocking vehicle checks if it is safe to do so and then performs the maneuver. This collaboration is beneficial for both vehicles in the long term (figure 6).

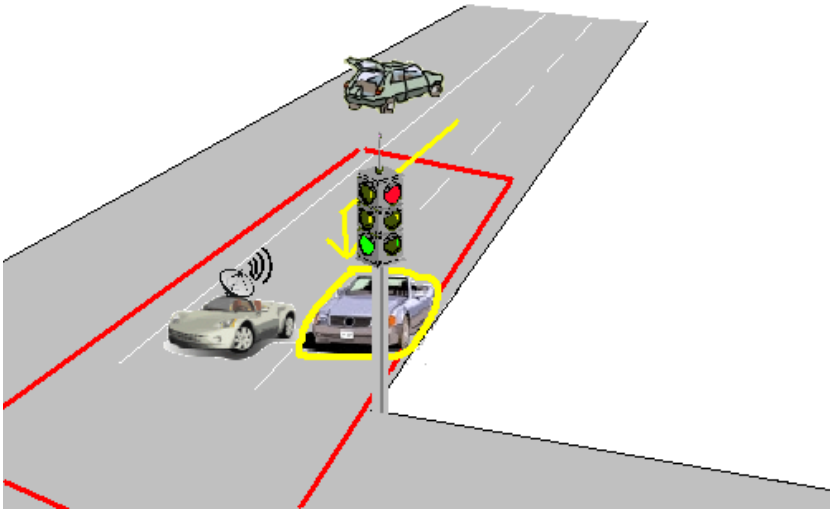


Figure 6:

- Although the vehicle has the right of way at the junction, it has received a broadcast event from an ambulance, requesting a clear path. The vehicle waits at the junction until the area is clear (figure 7).

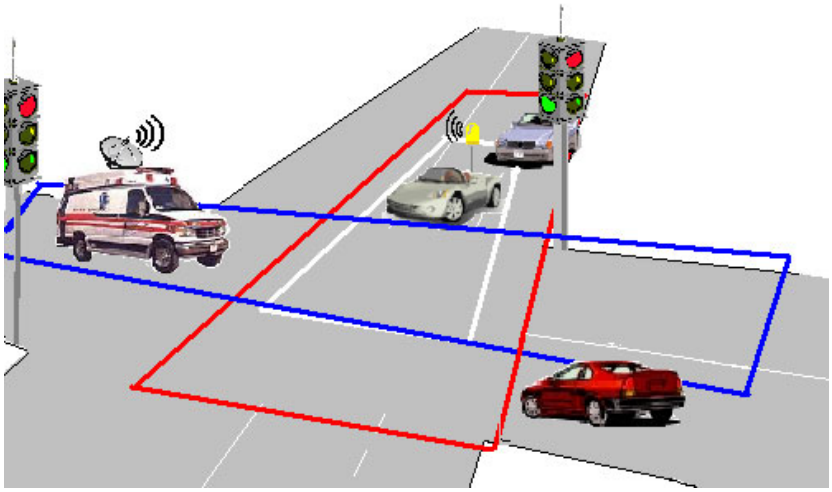


Figure 7:

- The ambulance passes and the vehicle detects that it is safe to resume its journey. It automatically alters its desired speed since it has changed from a two lane road to a single lane road (figure 8).

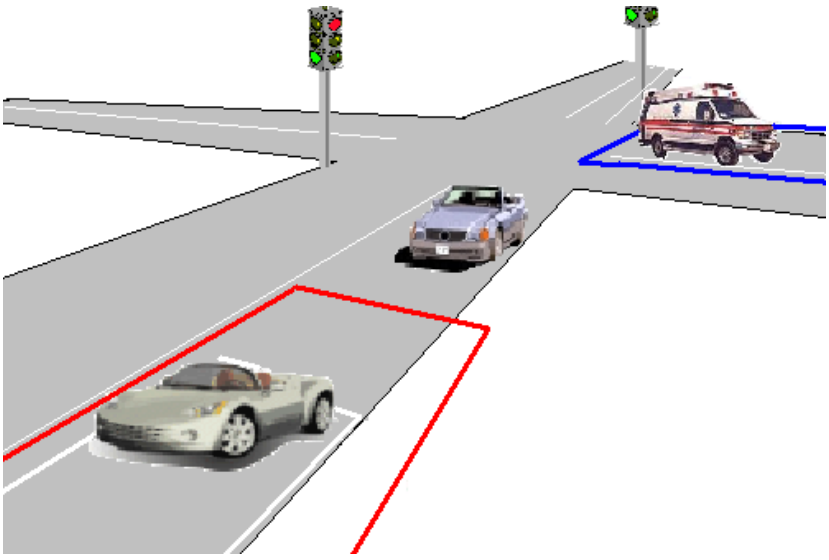


Figure 8:

5 Sentient Room Demo

The sentient room demo aims to build an intelligent environment using the sentient object paradigm. We have set the sentient room demo in a semi-public space the Innovative Interactions Laboratory (IIL) in computing department at Lancaster University. The main body of the IIL is the living room with a size of 7m*5.25m, and it is illustrated in Figure 9. The living room contains visual devices, e.g., cameras, plasma screens; audio devices, e.g., speakers, microphones; and conventional room devices, e.g., lights, air-conditioner.

Figure 9: The semi-public living room of IIL.

At the entrance of the room, an iris scanner and a library card reader (Figure 10) are installed, which can be used to identify the person who enters the room.

Figure 10: Iris scanner and library card reader at the entrance of IIL.

We have chosen to model the hardware devices as sentient sensor and actuator software objects, e.g., the iris scanner, plasma screen, environmental sensor, etc. We have engineered an event channel based on the TCD's STEAM and built a preliminary prototype – the personalized homepage launching service. This service consists of the iris scanner sensor, the plasma screen actuator, and a sentient object performing reasoning in the middle. The prototype works as the follows: the iris scanner sensor produces a recognized event in XML format when someone enters the room; the sentient object consumes this event, maps the user identity that is extracted from the recognized event content to his/her homepage URL, and produces a display event; the plasma screen actuator consumes the display event and launches Internet explorer to the specific URL (this scenario could easily be expanded to incorporate other actuations, such as controlling physical room attributes: temperature, light etc.).

This demonstrator contains the core sensor, actuator, and sentient object components of the CORTEX programming model. A key component missing from our demonstrator is the inference engine or controlling logic in the sentient object. We intend to put the inference engine into the sentient object, and then refine our proof-of-concept prototype – a more sophisticated homepage launching service. The focus of the prototype is to develop the contextual reasoning component within our sentient object prototype, in order to 'give intelligence' to the room, so that it can decide how to display homepage when there are multiple persons coming to the room.

Figure 11: How should the sentient room display the third person's homepage?

The room can have different possible ways on how to display multiple persons homepage: it can either display the homepage of the person who has the highest priority (the professor) or split the screen to display multiple homepages. No matter how the room decides to display the homepages, it has to have some decision making capability or intelligence, based on the limited contextual and sensor data available to it.

6 Demo of Framework for Testing Safety-Critical Sentient Applications

Several classes of sentient applications have *safety-critical* requirements. These requirements emerge from safety rules imposed by the environment and must be preserved by the system under any circumstance. Some examples of safety rules include keeping the temperature of a nuclear reactor under a threshold value, avoiding collisions between moving objects (e.g. planes, cars), etc. Some safety-critical systems have real-time requirements. In these systems, the computations must comply with timeliness constraints so that safety rules can be secured. Since failures of safety-critical systems can lead to catastrophic scenarios (e.g. human life loss, environmental disasters), this kind of systems must be highly dependable.

The correctness of safety-critical systems depends on their ability to secure safety rules imposed by the environment. These safety rules are commonly expressed as a set of requirements imposed on the behavior of both software and hardware components. Therefore, to test the correctness of these kind of systems it is often not sufficient to simply evaluate the software components. It is also necessary to deploy hardware devices in order to test their correct operation.

However, in large-scale sentient applications that involve the use of large amounts of expensive hardware (e.g. an application involving several vehicles approaching a road intersection), it is not possible to acquire all the necessary hardware devices to develop and test the applications because of cost reasons. Furthermore, since safety-critical requirements may also have to be preserved during the test phases, it is too risky to make these tests using "real" hardware. For example, testing *fly-by-wire* planes should not be done with real aircrafts. This line of reasoning is also applied in our everyday life. For instance, driving schools teach the basic driving principles to their students using driving emulators instead of real vehicles (the reason is obvious).

Thus being, it is obvious that using real hardware devices to test sentient applications may not be always possible. In order to address this issue we propose a software platform for the emulation of real environments, which can be used to test safety-critical, safety-related, or even money-critical CORTEX applications. This framework may be a very useful tool in the development of critical systems, in the sense that it complements all the other software engineering methodologies traditionally employed in this area (e.g. formal methods for the specification, development and verification of software and hardware systems, redundancy).

In what follows we describe the proposed emulator and a fictitious environment that was created for the purpose of demonstrating the emulator. Moreover, we also describe a distributed sentient application that controls a process in the mentioned environment. The application was implemented using *ATES* (*Adaptable Timed Event Service*) publishing/subscribing services (see deliverable WP3-D5).

The software emulator is illustrated in figure 12. There exists a central component that emulates a real environment (we call to this module *Controlled System Emulator*, or *CSE*). The environment emulated in this demonstration is rather simple. Nevertheless it can be as elaborated as necessary, depending on the application purposes. The fictitious environment consists of a bi-dimensional space delimited by walls in which four virtual controlled entities move at a certain speed and in several directions. Each entity is defined by four physical attributes: a position $< x, y >$ in

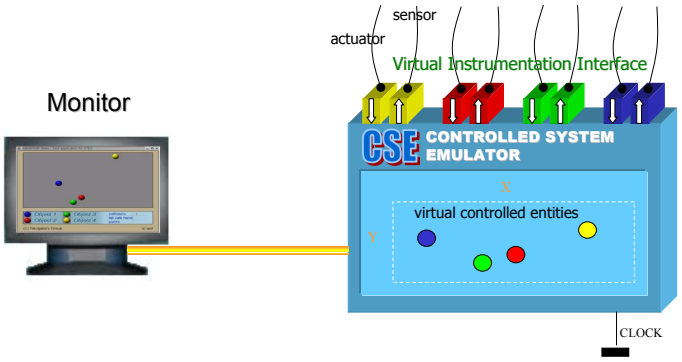


Figure 12: Emulation system.

the space, a shape, a speed and a direction. Physical laws such as friction or kinetic energy transfers are not modelled in this (environment) demonstration. Therefore, entities tend to keep their movements in a frictionless space unless they collide with a wall or with other moving entity, changing their direction in such a case.

The functioning of the emulator is engaged by a periodic clock signal with a known frequency. The environment emulated inside the *CSE* is updated at each clock tick, being the positions of all the controlled entities updated accordingly to their evolution rules, in this case, speeds and directions.

The emulation system has a local monitoring component that provides a human perceptible representation of the emulated environment. Periodically, the monitor gathers the state of the emulated environment from the *CSE* and displays a graphical representation of the latter to the user. It is worth to mention that the refresh rate of graphical frames in the monitoring component and the rate of update operations performed on the emulated environment (triggered by the clock) are unlikely to be the same. Firstly, when the *CSE* is clocked at high frequencies (e.g. 1000Hz) it is virtually impossible to enforce the graphical hardware to display all the states of the emulation. Moreover, the definitive restriction is the human inability to process visual information above a threshold frequency. Despite the impossibility of representing graphically all the states of the emulation, it is however guaranteed that all relevant events occurred in the emulated environment (e.g. collisions in our example) are notified to the user, either by providing counters of relevant events, or through suitable graphical representations. Figure 13 shows the monitor developed for this demonstration and the way in which collisions are notified to the user.

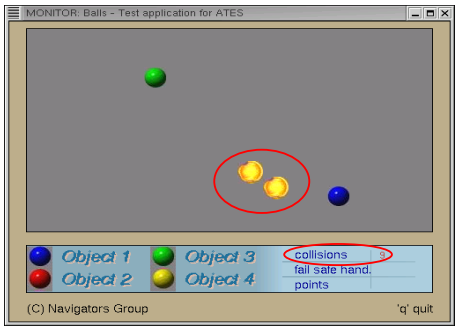


Figure 13: Monitoring component.

The *CSE* provides a virtual instrumentation interface composed of sensors and actuators to control applications. In our example, for each entity emulated inside the *CSE* there exists an associated sensor/actuator pair. A sensor allows applications to acquire the state of a controlled entity. The state of an entity is a triple containing its position, speed and direction. The actuators can be used by applications to change the movement (speed and/or direction) of the entities emulated inside the *CSE*[1].

Each sensor/actuator pair is locally made available to the controller application to allow distributed applications (i.e. CORTEX sentient applications) to access this instrumentation interface. In order to provide sensors with bounded errors towards the real state of the emulated environment, and actuators whose actions are timely propagated to the emulated environment, it is necessary to use communication links provided with the adequate timeliness to connect each sensor/actuator pair on a remote machine to the *CSE* central unit. At the bottom of figure 14 is depicted an internal view of the virtual instrumentation interface provided by the emulator.

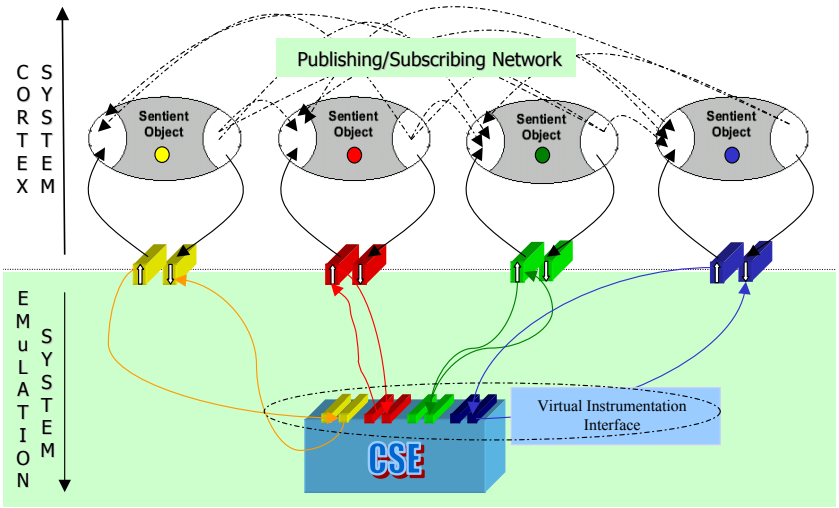


Figure 14: A CORTEX system with the emulator.

In this demonstration the entities emulated inside the *CSE* are shaped as four colored balls: red, green, blue and yellow balls. These balls emulate physical entities and prefigure real objects with similar dynamics. These can be cars, robots or even sub-atomic particles.

As it was mentioned above in this text, the balls (our virtual controlled entities) move in a bi-directional space changing their directions when collisions take place. Due to the absence of emulating functions for physical laws such as friction or energy transfers between objects when collisions take place, the balls tend to keep their speed constant (however, as it will be seen ahead, the speed can be changed by a control application). Initially, the balls start their movements at random speeds and directions.

A ball is a passive entity that moves in an uncontrolled manner unaware of its surrounding environment. However, the goal of the application described in

[1]For the sake of simplicity, we consider that a single sensing device provides the entire state of an emulated entity. The same is also true for actuators; a single actuator handles several types of commands.

the following paragraphs is to control the movement of the balls in order to avoid collisions among them, which is the safety rule that must be ensured by the system.

The infrastructure that was used to build this framework and demonstration is illustrated in figure 15. The emulator runs in a dedicated machine that executes a real-time version of the *Linux* operating system (*RTAI-Linux*). The *CSE* runs as a hard real-time task of *RTAI* kernel and is executed periodically (it is the clock signal of the *CSE*). The monitoring application is a *X11* application (user-level process) with a *GUI* that periodically acquires the state of the *CSE* (through an *RT-FIFO* device) and displays a graphical representation of the emulated space and balls.

Each CORTEX machine runs a sentient object that controls one of the emulated balls, using a virtual sensor and a virtual actuator as depicted in figure 14.

The instrumentation interface of the *CSE* is spread throughout the machines (CORTEX system nodes) deployed in this demonstration. Each sensor/actuator pair associated to a ball is locally made available to the respective sentient controller. There are user dedicated real-time communication channels in order to endow remote sensors and actuators with the necessary timeliness. For example, the sensor/actuator pair related to the yellow ball is deployed on the CORTEX node where the controller for this ball executes and is connected to the *CSE* through a dedicated real-time communication channel. Since sensor/actuator pairs may be deployed on different machines, a ball controller may not access (and should not, even if allowed to) other ball instrumentation interfaces.

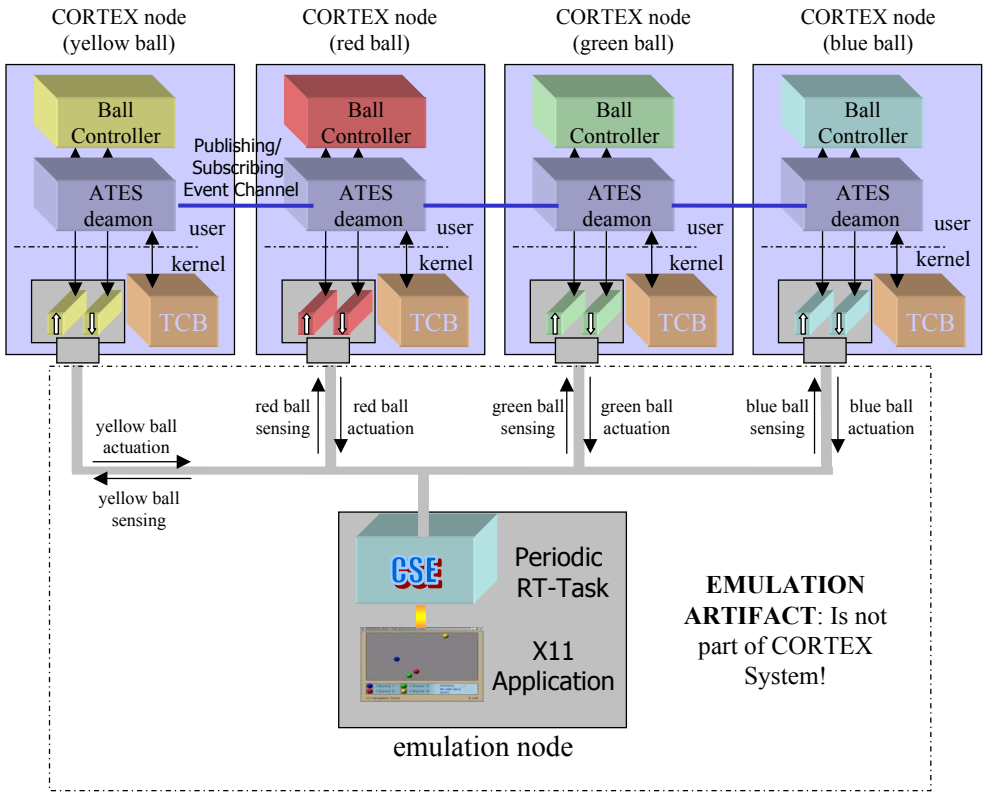


Figure 15: Application setup.

Sentient objects use *ATES* services to publish and subscribe information concerned with the control algorithm. A sentient object publishes the position, speed and direction of the ball that it controls, and subscribes the same information from

the other sentient objects in order to build a real-time image of the overall system. This real-time image will contain the position of every ball and will allow to decide if and how to actuate on the controlled ball. The details of the control algorithm are out of the scope of this document.

It is important to observe that this emulated scenario could be mapped in a real environment in which each ball could be materialized by a physical entity (e.g. a robot) with an embodied control unit (e.g. a micro-controller) and real sensors (e.g. *GPS*) and actuators (e.g. motor actuators).

In this application scenario, the most important requirement is to ensure that every ball controller has a consistent view (in time and space) of the environment, so that they can know each other's positions and so that collisions are avoided. Furthermore, given that: a) controllers periodically disseminate information containing the information of their controlled balls (position, speed and direction), and b) each controller must achieve a consistent view in time and space of other ball positions, there exist some timeliness, or real-time, requirements (network communications and local processing) that must be verified. As a matter of fact, in order to ensure that controllers only make decisions that preserve the safety of the overall system, it is necessary that, at every instant, each ball controller knows the position of all other ball with a known and bounded positioning error. To achieve that, a maximum ball speed must be defined and each controller must perform the following computations:

- disseminate the information of its controlled ball to the other ball controllers in a timely manner, that is, periodically and with a bounded communication latency. Based on the information received from other ball controllers, each controller should represent internally in a real-time image its perception of the surrounding environment;

- timely process the information when it is received from other controllers. Actuation in response to specific conditions (e.g. change the ball direction to avoid colliding with another one) must be also timely undertaken.

Temporal correctness of the execution of the above computations implies that the computational infrastructure where the application is running exhibits real-time properties. If this is not fulfilled, the application might not be able to secure the required safety property and therefore balls may collide.

Sentient objects use *ATES* publishing/subscribing services to disseminate the state of their controlled balls and to acquire the information related to the other balls. Although *ATES* services do not enforce real-time properties, ATES is built on top of a Timely Computing Base and therefore can use the TCB timing failure detection service. The application makes use of these TFD primitives in order to force the system to switch to a fail-safe state in a timely manner when timing failures occur, that is, when the expected real-time properties are not met by the infrastructure. The obvious fail-safety procedure in this case is to force the affected balls to stop.

In mobile environments, where devices may travel through several networks, it is difficult to secure real-time properties for the underlying infrastructures. Moreover, mobile environments are inherently dynamic (e.g. several devices entering and leaving a network) and therefore timing properties of the services provided by a network (that is, the supported *Quality-of-Service (QoS)* may oscillate over time.

Although the safety of the system is ensured by using the timing failure detection primitives of *ATES*, it is however desirable to minimize the activation of fail-safety procedures. In fact, it is important to ensure the progress of system applications even in situations of degraded timeliness of the underlying infrastructure. Therefore, some mechanisms must be employed to prevent systems from being blocked on fail-safe states, to which they switch because of incorrect assumptions about system timing properties that lead to recurrent timing failures.

Consider, for instance, that the balls were cars and that the fail-safety procedure consisted in forcing the cars to break suddenly. In this scenario, it would be desirable to reduce the activation of fail-safety procedures to the minimum, to preserve the breaking engine of the car and, of course, to keep passengers happy! On the other hand, since the objective is to keep moving as fast as possible, this raises a tradeoff with the need to minimize the activation of fail-safety procedures.

ATES provides extensions for dependable *QoS* adaption, giving feedback to applications of the changing conditions of the infrastructure. The controllers use these *QoS* adaptation extensions to update their timing requirements to the available QoS. Moreover, they adjust the speed of the balls (issuing proper commands to the actuators) according to the changing timing requirements and thus minimizing the activation of fail-safety procedures.

In this demonstration it is possible to observe the following issues:

- the effect of timing failures, leading to collisions between balls;
- the significance of being able to detect timing failures in a timely manner, forcing the balls to stop quickly in order to avoid collisions;
- the importance of the adaptation procedures, allowing to smooth the movement of the balls.

Furthermore, in order to observe the behavior of the application under several timeliness conditions, we have developed a module that allows us to inject artificial timing failures (possibly omissions) on the *ATES* publishing/subscribing channel.

References

[1] IEEE Standard 802.11b-1999 (supplement to ANSI/IEEE Standard 802.11, 1999 edition), 1999.